$\operatorname{CLP}(\mathcal{R})$ and Some Electrical Engineering Problems^{*}

Nevin Heintze Spiro Michaylov

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A. Peter Stuckey IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.

November 25, 1991

Abstract

The Constraint Logic Programming Scheme defines a class of languages designed for programming with constraints using a logic programming approach. These languages are soundly based on a unified framework of formal semantics. In particular, as an instance of this scheme with real arithmetic constraints, the $\text{CLP}(\mathcal{R})$ language facilitates and encourages a concise and declarative style of programming for problems involving a mix of numeric and non-numeric computation.

In this paper we illustrate the practical applicability of $\text{CLP}(\mathcal{R})$ with examples of programs to solve electrical engineering problems. This field is particularly rich in problems that are complex and largely numeric, enabling us to demonstrate a number of the unique features of $\text{CLP}(\mathcal{R})$. A detailed look at some of the more important programming techniques highlights the ability of $\text{CLP}(\mathcal{R})$ to support well-known, powerful techniques from constraint programming. Our thesis is that $\text{CLP}(\mathcal{R})$ is an embodiment of these techniques in a language that is more general, elegant and versatile than the earlier languages, and yet is practical.

1 Introduction

The Constraint Logic Programming Scheme [8] defines a family of declarative and formallybased languages for reasoning about constraints. An instance of this scheme, the $CLP(\mathcal{R})$ language [6, 9], deals particularly with arithmetic constraints. In this paper we demonstrate the features of $CLP(\mathcal{R})$ with some problems from electrical engineering. This important problem domain, requiring a variety of different solving techniques, provided the driving examples for the pioneering work on constraint programming [3, 15, 18]. The complexity of the systems arising in electrical engineering problems, together with the large number of arithmetic constraints typically involved, made this problem domain an obvious candidate for the application of constraint programming languages.

^{*}To appear in the Journal of Automated Reasoning. An earlier version of this paper appeared in the proceedings of the 4th International Conference on Logic Programming, Melbourne, May 1987. Much of this work was carried out while the authors were at Monash University, Melbourne, Australia.

It is not our intention in this paper to propose new techniques for solving these problems, but rather to show how effectively the well-known techniques developed by researchers in constraint programming can be used in conjunction with the $\text{CLP}(\mathcal{R})$ language. In fact, while these techniques could previously be applied only in special purpose hand-crafted code, here we incorporate them in simple programs within a cleaner and more general programming framework. Furthermore, while $\text{CLP}(\mathcal{R})$ cannot claim to solve all the problems of constraint programming, it is more easily applicable to a much larger class of problems than any of its predecessors. Significantly, our experience has also indicated that the resulting programs are usually practical — often surprisingly so.

The remainder of the paper is organized as follows. In section 2 we give a brief description of the $\text{CLP}(\mathcal{R})$ language and system. Then in section 3 we examine some approaches to programming with constraints, comparing the $\text{CLP}(\mathcal{R})$ approach with that of other constraint languages. In section 4 we discuss software for the analysis of circuits. The two major examples are steady-state analysis of RLC circuits, and synthesis and analysis of transistor amplifier circuits. Section 5 describes software for the simulation of digital filtering circuits. Finally, in section 6 we discuss the analysis of electro-magnetic fields.

2 The $CLP(\mathcal{R})$ Language and System

We give a very brief description of the $\operatorname{CLP}(\mathcal{R})$ language. For more details, see [6, 7, 9]. Arithmetic terms are constructed from real constants, variables, +, -, *, /, sin, cos, tan, pow where all of these symbols have the usual meanings and parentheses may be used in the usual way to resolve ambiguity. Constraints are built up from arithmetic terms using the binary relation symbols =, \geq , \leq , >, <. For example 1.234 + X < Y and $X + Y * (\sin(T) - 1) = \tan(Z)$ are constraints. Any variable that appears in an arithmetic term is said to be an arithmetic variable, and cannot take a non-arithmetic value. Terms are constructed from variables, arithmetic terms and uninterpreted functors. For example, (X + Y)/4 and g(22, h(4 * (Y + X))) are terms, while f(X) + g(X) and a - 3 are not. Atoms are of the form $p(t_1, \dots, t_n)$ where p is an n-ary predicate symbol and the t_i are terms. Finally a program is a finite collection of rules, each of the form:

 $H := B_1, \cdots, B_m$

where H is an atom and each B_i is either a constraint or an atom.

For example, the following single-rule program models the relationship between two complex numbers and their product, where each complex number, X + iY, is represented as c(X, Y).

The operational model for $CLP(\mathcal{R})$ is an obvious generalization of that for PROLOG. Briefly, a goal consists of a number of solvable constraints and a number of atoms. A derivation step consists of first matching a selected atom with the head of an input rule (whose variables have been suitably renamed). This matching in general generates new constraints. The solvability of the collection of constraints consisting of (a) the constraints in the previous goal, (b) those new constraints generated by the head matching, and (c) the constraints in the body of the rule, is then determined. If (a-c) is solvable, a new (derived) goal is constructed, consisting of this new collection of (solvable) constraints and the atoms of the previous goal, with the exception that the selected goal atom is replaced by the atoms in the body of the input rule. We refer the reader to [6, 7] for more details.

Consider the complex multiplication program given previously. Any of the following goals will return a unique answer. The first goal asks for the product of two complex numbers, while the other two ask for the result when one complex number is divided by another.

?- c_mult(c(1, 1), c(2, 2), Z), ?- c_mult(c(1, 1), Y, c(0, 4)), ?- c_mult(X, c(2, 2), c(0, 4)),

The examples in this paper were developed on the $\text{CLP}(\mathcal{R})$ system of [7, 9]. This system is an approximation to the operational model of the $\text{CLP}(\mathcal{R})$ language because: (a) the atom selection rule is "left-to-right" (and so the system is incomplete in the same sense that a PROLOG system is incomplete), (b) constraints involving non-arithmetic terms are solved by a unification algorithm which omits the occurs check (and so the system is unsound in the same sense that a PROLOG system is unsound), (c) a floating point representation is used for arithmetic, and (d) any non-linear arithmetic constraints are assumed to be satisfiable, and are "delayed" from consideration — such a constraint may become linear at some time later in a computation¹, at which stage it will be considered for satisfiability.

Point (d) represents a compromise between generality and efficiency. It restricts consideration to a class of constraints which can be solved efficiently. However, by relaxing the condition of satisfiability for non-linear constraints, rather than disallowing them altogether, the $\text{CLP}(\mathcal{R})$ system admits an important additional class of programs.

Briefly, the $\text{CLP}(\mathcal{R})$ system solves constraints in the following way. Linear arithmetic constraints (both equalities and inequalities) are maintained in a solved form. As each linear constraint is added, it is checked for satisfiability with the previous constraints (already in solved form). If the system is satisfiable, the new constraint is incorporated to generate a new solved form. A delayed (non-linear) constraint is woken and added to the solved form when a sufficient number of its variables have been determined to make it linear. For example in the program:

```
hyp(X, Y, pow(X*X + Y*Y, 0.5)).
available(X,Y):- X = Y.
?- X + Y = 10, Z <= 8, hyp(X,Y,Z), available(X,Y).</pre>
```

execution proceeds by collecting the two linear constraints, X + Y = 10, and $Z \le 8$; clearly jointly satisfiable. Next, the matching of the goal atom hyp(X,Y,Z) and the head of the rule hyp(X, Y, pow(X*X + Y*Y, 0.5)) gives Z = pow(X*X+Y*Y, 0.5). This constraint is delayed since it is not linear. The next constraint collected is X = Y, and this together with X + Y = 10, determine the values X = 5, and Y = 5. Now the delayed constraint is woken and Z = pow(5*5+5*5,0.5) = 7.07 is added to the linear constraints. Since the linear constraints are satisfiable the goal succeeds.

¹This may happen when some of the variables contained in a non-linear constraint become ground.

3 Comparison with Other Constraint Languages

A constraint programming language allows reasoning in some domain through the use of constraints over objects in that domain. For example, consider the problem of modeling an electrical system consisting of a collection of components of different types. To program this in a constraint language, we could write down constraints describing each of the types of component. Further constraints would be used to describe the interconnection of these components.

An important language question concerns the treatment of constraints appearing in a program – do they represent constraints as such, or are they templates for constructing constraints at runtime? We will refer to the former as *static* constraints, and to the latter as *dynamic* constraints, and to runtime copies of such constraints as constraint *instances*. Consider the electrical modeling problem again. If we only have static constraints, then constraints will have to be written for each component. However the ability to program with dynamic constraints allows us to define a constraint "template" for each type of component, and then for each component instance, to construct a copy (or instance) of the "template", possibly with extra values filled in. This leads to a notion of collection of (instances of) constraints at runtime.

In THINGLAB [1] constraints are static. Steele's language [16] is slightly more general, as it allows programs to contain macro constraint definitions. However, it disallows recursion, so that the number of instances of each constraint, and the relationship between the instances, is fixed. In $\text{CLP}(\mathcal{R})$ the use of recursive rule application allows the number of instances of each constraint, and the relationship between those instances, to be determined at run-time.

A second question is whether properties of the constraint system collected so far may influence the future collection of constraints. That is, whether the structure of the constraint system constructed at runtime can be influenced by properties of the partially constructed constraint system. Most constraint languages do not support this technique. However in $CLP(\mathcal{R})$ it is inherited directly from the operational model, since at each stage in a derivation the constraint system collected so far is checked for satisfiability. Hence constraints arising from head matching or appearing in the body of a rule may be used to "check" various properties of the current constraint system, before the rule is used to replace the selected goal atom.

Additionally, $\text{CLP}(\mathcal{R})$ provides the usual PROLOG-like function symbols for dealing with aggregate data such as lists and records. This not only means that $\text{CLP}(\mathcal{R})$ subsumes PROLOG, providing support for symbolic computation, but it also provides a simple way to store data needed to set up constraint instances and to keep the results of solving these constraints.

Much of the programming methodology described in this paper, and support for many of the constraint programming techniques discussed, rely heavily on these features.

Execution of a constraint program involves constructing the constraint system (or systems) dictated by the program and determining if the system is satisfiable [as well as constructing a representation of the answer constraints]. There is an important distinction between the constraint programming language, and the algorithm used to solve the constraints — the *constraint solver*.

There are many approaches to solving a given set of (numerical) constraints. One



Figure 1: Simple Resistive Circuit

approach is *local propagation*. This is the sole method for solving arithmetic constraints in the work of Steele [16, 17], in PROLOG systems including MU-Prolog [12] and NU-Prolog [19], and elsewhere. A system of constraints is solved by local propagation if all the variables in the system become determined after a finite number of local propagation steps. A *local propagation step* occurs when a constraint has a sufficient number of determined variables for some of its other variables to be determined. These newly determined variables may then precipitate further local propagation steps in other constraints. For example consider the simple electrical circuit in Figure 1.

This circuit can be modeled by the constraints

$$I_T = I_1 + I_2$$
$$V = I_1 * R_1$$
$$V = I_2 * R_2$$

Suppose that we are given values for V, R_1 and I_T , then all unknowns can be determined using local propagation as follows:

$$\begin{array}{rcccc} I_1 & \leftarrow & V/R_1 \\ I_2 & \leftarrow & I_T - I_1 \\ R_2 & \leftarrow & V/I_2 \end{array}$$

where \leftarrow denotes assignment. On the other hand if we are given R_1 , R_2 and I_T , it is not possible to order the equations so that they can be solved by this simple evaluation and assignment method since there are cyclic interdependencies between the variables; the system cannot be solved by local propagation. Such situations occur frequently during the solving of electrical circuit problems. This prompted Sussman & Stallman [18] to introduce a special heuristic to handle the voltage divider law in their circuit analysis package – however this approach is limited to solving a few special cases.

In general, solving systems of constraints involving cyclic interdependencies requires the use of more powerful techniques than local propagation. Among the approaches taken in the literature are relaxation methods [1], and linear equation solving with user assistance [20]. To solve general arithmetic constraints, which may be nonlinear, still more powerful techniques are required, such as those of symbolic algebra packages like REDUCE [14] and MACSYMA [11]. Though the power and generality of such constraint solvers is enviable, they are usually too slow to be incorporated in a general purpose programming

language. For example a critical issue in the constraint systems EL/ARS [15, 18] and SYN [3] which used MACSYMA as the basic constraint solver, was the development of intelligent backtracking to reduce the amount of work being performed by MACSYMA. In the $CLP(\mathcal{R})$ system, linear constraints are solved directly and non-linear constraints are delayed as described in the previous section. This provides more power than the methods of local propagation, but avoids incurring the stiff computation expense involved in the more powerful general-purpose symbolic algebra constraint solvers.

While it is possible to augment any language with constraint facilities, an important issue is how the underlying language interacts with the constraint facilities. In some constraint languages this interaction is clumsy, requiring the user to give a great deal of information about how constraints are to be collected and solved (for example [5, 10, 20]). The constraint logic programming scheme, in contrast, defines a clean interaction between the underlying logic programming framework and constraint facilities. Other languages in this group include Prolog III [2] and CHIP [4]. Both languages include equations and inequalities over rational numbers (in addition to other domains). However in these languages the programmer is responsible for ensuring that nonlinear expressions have become linear by the time they are encountered during program execution, thus compromising the declarative reading of the constraints. Finally, all arithmetic in both languages is based on an exact representation of rational numbers, and it has yet to be demonstrated that this representation is efficient enough for the kinds of numerically intensive applications considered in this paper.

We conclude this section with a $CLP(\mathcal{R})$ program which illustrates the use of dynamic constraints, data structures, and the use of satisfiability requirements to control search. Although this example is very simple, it would be extremely difficult to program it in the earlier constraint languages, and would be impossible in many of them. The program models any circuit component using a piecewise linear model.

```
linpiece(X,[piece(C1,C2,F1,F2)|Ps]) :-
C1 < X,
X <= C2,
F1 = F2.
linpiece(X,[P|Ps]) :-
linpiece(X,Ps).
```

Intuitively the program checks whether the value X falls within the limits C1 and C2 and if so creates an equation F1 = F2. The simplest circuit component can be modeled as follows

```
resistor(V, I, R):-
    linpiece(V, [piece(_,_, V, I * R)]).
```

Diode behavior can be modeled using the piecewise linear approximation of Figure 2. The three linear pieces correspond to the three areas of operation of the diode — reverse breakdown, reverse biased and forward biased. Non-determinism and backtracking together take care of the question of which state the diode may be in for a particular circuit.



Figure 2: Piecewise linear diode model

The following goal determines the behavior of the circuit in Figure 3 for circuit values $V = 5 \text{ volts}, R_1 = 100\Omega, R_2 = 50\Omega, R_3 = 50\Omega, R_4 = 100\Omega$

```
?- V = 5, R1 = 100, R2 = 50, R3 = 50, R4 = 100,
resistor(V-A, I1, R1),
resistor(A, I2, R2),
resistor(V-B, I3, R3),
resistor(B, I4, R4),
diode(B-A, I5),
I1 + I5 = I2, I3 = I5 + I4.
```

4 Analysis and Synthesis of Analogue Circuits

Before considering examples of the analysis and synthesis of analogue circuits in $\text{CLP}(\mathcal{R})$, we will briefly sketch a broad methodology for such programming. In general the constraints of such a system can be viewed in terms of a hierarchy. *Leaf constraints* describe the relationship between variables at a sub-system or "local" level, for example Ohm's law for a resistor in a circuit. *Parent constraints* describe the interaction between these sub-systems, for example the use of Kirchoff's current law in node analysis, which states that the sum of currents flowing into a node is zero. This distinction assists the writing of programs whose hierarchical structure reflects that of the problem to be solved. Such programs are easier to understand and reason about. In this programming methodology, leaf constraints are usually encapsulated within a single rule, while parent constraints are programmed as a single rule which combines a number of program modules.



Figure 3: Resistive circuit with diode

4.1 Analysis of Steady-State RLC Circuits

We first consider the application of this approach to the analysis of sinusoidal steady-state RLC circuits. We represent (sinusoidal) voltages and currents in the circuit as phasors using complex numbers, where c(X, Y) is used to represent X + iY. For example the inductor and capacitor have voltage (V) – current (I) relationships $V = I(i\omega L)$ and $V = I/(i\omega C)$ respectively, where L is inductance, C is capacitance and ω is angular frequency. These devices may be modeled by the following rules:

```
inductor(V, I, L, Omega) :-
    c_mult(I, c(0, Omega*L), V).
capacitor(V, I, C, Omega) :-
    c_mult(V, c(0, Omega*C), I).
```

To connect networks of these components together, we have to satisfy the various conservation laws at the interface of the components. This can be done by ensuring that:

- all component terminals that are connected together are at the same voltage, and
- for each node, the sum of the currents flowing into the node is zero.

Appendix I contains a general package for solving arbitrary sinusoidal steady-state RLC circuits which incorporates these ideas. The hierarchy of constraints begins with operations for complex number arithmetic, then circuit element internal relationships in terms of complex arithmetic operations, and finally circuit structure relationships in terms of circuit element variables. The use of this approach has resulted in a program which is structurally similar to the problem to be solved. It has also aided reasoning about the program. In particular it has helped to ensure that all information about the circuit has been captured and that important constraints have not been inadvertently omitted.

The package is run using the predicate circuit_solve() with four arguments. They are respectively: the source frequency, circuit representation, the nodes at ground potential and the nodes whose values are to be printed after the analysis. The package can also be



Figure 4: Use of the general package to solve a DC circuit

used for D.C. analysis by setting the frequency and imaginary parts of quantities to zero. This is illustrated by the following goal which analyzes the circuit of Figure 4.

```
? - W = 0,
        Vs = 10,
        R1 = 100,
        R2 = 50,
        circuit_solve(W,
                         Г
                         [voltage_source,v1,c(Vs,0),[n1,ground]],
                         [resistor, r1, R1, [n1, n2]],
                         [resistor, r2, R2, [n2, ground]],
                         [diode, d1, in914, [n2, ground]]
                         ],
                         [ground],
                         [n2]
                         ).
Output is:
    COMPONENT CONNECTIONS TO NODE n2
    resistor r1: 100 Ohms
    Node n2 Voltage c(0.60082, 0) Current c(-0.0939918, 0)
    resistor r2: 50 Ohms
    Node n2 Voltage c(0.60082, 0) Current c(0.0120164, 0)
```

The second example shows a larger RLC circuit (Figure 5). The goal shown generates 300 equations and is run by the $CLP(\mathcal{R})$ system in 0.5 seconds on a Pyramid 98X.

Node n2 Voltage c(0.60082, 0) Current c(0.0819754, 0)

?- W = 100, Vs = 10, Tr1 = 5, Tr2 = 0.2, R1 = 200,

diode d1: type in914



Figure 5: RLC Circuit

```
R2 = 1000,
R3 = 50,
R4 = 30,
C1 = 0.05
L1 = 0.005,
circuit_solve(W,
        Г
        [voltage_source, v1, c(Vs,0),[in, ground1]],
        [resistor, r1, R1, [in, n1]],
        [transformer, t1, Tr1, [n1, ground1, n2, ground2]],
        [resistor, r2, R2, [n2, n3]],
        [capacitor, c1, C1, [n3, n4]],
        [resistor, r3, R3, [n4, ground2]],
        [transformer, t2, Tr2, [n4, ground2, out, ground3]],
        [resistor, r4, R4, [out, ground3]]
        [inductor, 11, L1, [out, ground3]]
        ],
        [ground1, ground2, ground3],
        [out]).
```

Output is:

```
COMPONENT CONNECTIONS TO NODE out
transformer t2: ratio of 0.2
Node out V c(3.34983e-06, 0.0001984) C c(-0.0003968 8.78204e-08)
resistor r4: 30 Ohms
Node out V c(3.34983e-06, 0.0001984) C c(1.11661e-07,6.61185e-06)
inductor l1: 0.005 Henry
Node out V c(3.34983e-06, 0.0001984) C c(0.0003967, -6.69967e-06)
```

4.2 Transistor Circuits: Analysis and Design

While the voltage-current behavior of the transistor is highly non-linear, it can be modeled for most practical purposes using piecewise linear techniques. We will illustrate this approach through the analysis and design of amplifier circuits using the bipolar junction transistor. This approach is equally applicable to other types of circuit and other types of transistor. Typical engineering methods for the design of transistor circuits are iterative and guided by heuristics. This involves estimating key circuit parameters, computing the remaining unknown values, and then making corrections to the original estimates. The programs we present use a similar approach. However, we replace iteration by backtracking.

The design of transistor amplifiers falls into three stages. First, the basic form of the amplifier is selected (for example: common-base, emitter-follower). Second, the biasing circuitry is designed to ensure that the transistor will operate in its active mode within the expected range of inputs to the circuit. Design requirements include insensitivity to transistor parameters such as common-emitter current gain, and temperature. Third, the requirements of the small signal operation of the amplifier are satisfied. These requirements may include high input resistance, low output resistance and predictable gain.

The package presented in appendix II provides both simple D.C. analysis for biasing and digital circuitry, and small signal analysis for transistor amplifiers. The transistor is modeled by three modes of operation: *active, saturated* and *cutoff.* For amplifier circuits we are primarily interested in the active mode of operation, while for digital circuits the saturated and cutoff modes become important. The following rules give the D.C. properties of an npn transistor (Figure 6) in the three modes. The variables *Beta*, *Vbe* and *Vcesat* are device parameters; *Vx* and *Ix* are respectively voltages and currents, where *x* ranges over *b* (base), *e* (emitter), *c* (collector).

```
transistor_dc(active, npn, Beta, Vbe, Vcesat,
             Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb = Ve + Vbe,
        Vc >= Vb, Ib >= 0,
        Ic = Beta*Ib,
        Ie = Ic + Ib.
transistor_dc(saturated, npn, Beta, Vbe, Vcesat,
             Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb = Ve + Vbe,
        Vc = Ve + Vcesat,
        Ib >= 0, Ic >= 0,
        Ie = Ic + Ib.
transistor_dc(cutoff, npn, Beta, Vbe, Vcesat,
             Vb, Vc, Ve, Ib, Ic, Ie) :-
        Vb < Ve + Vbe,
        Ib = 0,
        Ic = 0,
        Ie = 0.
```

With the addition of the previously defined rule for modeling the (D.C.) properties of resistors (section 3), we are able to analyze some simple circuits. For example the simple biasing circuit shown in Figure 7 may be analyzed using the following goal:

```
?- resistor(15 - Vb, I1, 100),
    resistor(-Vb, I2, 50),
    I1 + I2 = Ib,
    transistor(State, npn, 100, 0.7, 0.3,
        Vb, Vc, Ve, Ib, Ic, Ie),
    resistor(Ve, Ie, 3),
    resistor(15 - Vc, Ic, 5).
```



I

T

Figure 6: An NPN type transistor



Figure 7: Biasing Circuit for NPN transistor

The full program in appendix II is an extension of these rules to allow the D.C. analysis of more general transistor circuits, which may include capacitors and diodes. This program is written to reflect the structure of the problem by defining component voltage-current relationships at one level, and component connections, at a higher level, in terms of these voltage/current parameters. The circuit structure is defined by listing the components and their connections. For example, the following goal again analyzes the circuit of Figure 6.

Output is:

State = active

A number of different types of problems can be solved with this package. For example the following goal does not specify the component values, but instead constrains them to lie within certain ranges.

Output is:

```
State = active
R1 = 50
R2 = 27
```

Finally we can re-employ the analysis program as a design program. This can be achieved by constraining the design parameters, choosing a circuit template, and co-routining the search for suitable components with a circuit analysis – the *test and generate* technique. After a circuit template has been chosen, the circuit analysis rules are used to set up the appropriate constraints, and then the search for components takes place. When values are chosen for components, an inappropriate choice will often cause the system of constraints to become unsatisfiable immediately, leading to backtracking (assuming that the determination of unsatisfiability does not rely on the consideration of non-linear constraints). In this way we avoid an exhaustive search to find component values that satisfy the design constraints. A goal for designing a positive gain transistor amplifier is shown below.

```
?- Vcc = 15, Stability < 0.5, Gain > 0.5,
Inresistance >= 25, Outresistance <= 0.5,
full_analysis(Vcc, _, Circuit, _, _, Type, Stability,
Gain, Inresistance, Outresistance).
```



Figure 8: High frequency filter

Output is:

This result is obtained in 46 seconds on an IBM RT PC (APC), after solving over seven thousand linear constraints. The approach of co-routining constraint satisfaction with the search for components results in a search space pruning of two orders of magnitude and the total time taken is reduced by more than an order of magnitude over the exhaustive search approach.

5 Digital Signal Flow

We concentrate in this section on the simulation of linear shift-invariant digital systems with time as the independent variable [13]. We represent digital systems in terms of linear signalflow graphs. A signal flow graph is a collection of nodes and directed branches. Associated with each node is a variable which is the signal value at that node. Source nodes have their signal value determined at each stage by some input signal. Branches are either *multiplier* branches or *delay* branches. A multiplier branch multiplies the signal at its input end by its labeling coefficient to give its output; a delay branch has as output its input at the previous time step. The signals at each node in the signal flow graph are such that the signals in any outgoing branches are equal to the sum of the signals in the incoming branches (the *summing constraint*). As an example, consider the following digital filter:

We simulate the signal-flow graph by computing the signal values at the nodes at successive time steps, starting from some given initial values. This is achieved by collecting branch equations and summing constraints for each node. This, along with the input signal



Figure 9: Input and output for filter

values to the source nodes, is sufficient to determine the signal values at each node for each time step.

The signal-flow graph simulator appearing in appendix III is called using the predicate flow() with a description of the graph as the first argument and the name of the node where the value is to be printed as the second argument. The following goal describes the low pass filter of Figure 8.

```
?- flow([
        [source, in, n1],
        [delay, n1, n2],
        [delay, n2, n3],
        [delay, n3, n4],
        [delay, n4, n5],
        [coeff(0.2), n1, n6],
        [coeff(0.2), n3, n6],
        [coeff(0.2), n4, n6],
        [coeff(0.2), n5, n6]
        ],
        n6).
```

Figure 9 shows the output from the program in graphical form corresponding to the accompanying square wave input.

6 Electro-Magnetic Field Analysis

Often it is undesirable or impractical to analyze electrical systems in terms of lumped circuit components. In many cases an analysis using electro-magnetic field theory is required, typically involving the solution of partial differential equations subject to some boundary or initial conditions. A simple way to solve these problems is to use a finite difference approximation. We will consider the five-point approximation to Laplace's equation known as



$$\frac{u_O(ap+bq)}{apbq} = \frac{u_A}{a(a+p)} + \frac{a_B}{b(b+q)} + \frac{u_P}{p(p+a)} + \frac{u_Q}{q(q+b)}$$

Figure 10: Liebmann's 5-point approximation to Laplace's equation

Liebmann's method described in Figure 10, which can be used to solve Dirichlet, Neumann and mixed boundary value problems. The finite difference equations are the leaf constraints — they apply to a neighborhood of points. The complete finite difference problem is a parent constraint arising from the overlapping of these neighborhoods.

The following is a simple program to solve the Dirichlet problem for Laplace's equation in two dimensions.

```
laplace([_,_]).
    laplace([H1,H2,H3|T]):-
             laplace_vec(H1,H2,H3),
             laplace([H2,H3|T]).
    laplace_vec([_,_],[_,_],[_,_]).
    laplace_vec([TL,T,TR|T1],[ML,M,MR|T2],[BL,B,BR|T3]):-
             B + T + ML + MR - 4*M = 0,
             laplace_vec([T,TR|T1],[M,MR|T2],[B,BR|T3]).
?- laplace([[0,
                         Ο,
                              Ο,
                                    Ο,
                                               0],
                   0,
                                         Ο,
             [100, _,
                                         _, 100],
                         _ ,
                              _,
                                    _ ,
             [100, _,
                                         _, 100],
                        _,
                              _,
                                    _ ,
             [100, _,
                                            100],
                              _ ,
                                         _ ,
                         _ ,
                                    _ ,
             [100, _,
                                         _, 100],
                         _ ,
                              _,
                                   _ ,
             [100, _,
                                         _, 100],
                         _,
                              _ ,
                                    _ ,
             [100, 100, 100, 100, 100, 100, 100]]).
```



Figure 11: A 2-dimensional boundary value problem

Output is:

[[0, 0, 0, 0, 0, 0, 0, 0], [100, 53.1, 37.1, 33.1, 37.1, 53.1, 100], [100, 75.4, 62.1, 58.1, 62.1, 75.4, 100], [100, 86.5, 77.9, 75.0, 77.9, 86.5, 100], [100, 92.8, 87.9, 86.2, 87.9, 92.8, 100], [100, 96.9, 94.6, 93.9, 94.6, 96.9, 100], [100, 100, 100, 100, 100, 100, 100]]

The region of interest is represented as a matrix (list of lists) of anonymous variables, with constants at the edge specifying the boundary values. Constraints (finite difference equations) are collected at each point of the matrix by "sliding" a "window" of 9 points across the matrix and referencing only the variables in that window. The important point of this program is the simplicity with which the equations are collected.

The central methods of this program are expanded in a more general package to allow the user to describe the region of interest and the boundary conditions more concisely but still declaratively. The form of the goal, which is described below, is generalized to cope with Neumann and mixed boundary value problems. It can also deal with irregular boundaries. Figure 11 shows a 2-dimensional region enclosed on 3 sides by a barrier at 0 potential, and from one side by a barrier at potential of 100 volts. We may express this as the goal

Output is:

At (20,20) = 55.78 At (20,40) = 42.15

where the first argument describes the vertices of the region of interest together with the boundary values on the lines joining them, the second argument is the grid separation, and the third argument is a list of points at which the value of the potential is required. We could modify the program to provide for non-constant boundary values: they may be specified as bv(top) instead of b(0), in conjunction with the rule

boundary(top,X,Y,X*X+Y*Y).

added to the program by the user to specify that the potential at the top of the boundary depends on the distance from the bottom left hand corner, where the two axes are X and Y. The program represents the finite difference equations concisely, and utilizes the constraint collection mechanism of $CLP(\mathcal{R})$ to solve the problem in a natural way.

Conclusion

We have considered the application of $\text{CLP}(\mathcal{R})$ to a range of electrical engineering problems — RLC circuit analysis, transistor circuit design, digital signal flow and electro-magnetic field analysis — which require symbolic reasoning as well as significant numerical computation. Most of the programs in this paper were originally written to investigate the programming techniques supported by the $\text{CLP}(\mathcal{R})$ language and the practicality of the $\text{CLP}(\mathcal{R})$ system. Our experience has indicated that $\text{CLP}(\mathcal{R})$ naturally supports problem solving techniques from the early work on constraint programming languages. Moreover, $\text{CLP}(\mathcal{R})$ is a more general, elegant and versatile language, and yet is practical.

Acknowledgements

We are grateful to Joxan Jaffar and Jean-Louis Lassez as well as the referees for their comments on earlier versions of this paper.

References

- Borning, A. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", ACM Transactions on Programming Languages and Systems, Vol 3, No 4, October 1981, pp 252–387.
- [2] Colmerauer, A. "Opening the Prolog III Universe", Byte, 12 (9), August 1987.
- [3] De Kleer, J. and Sussman, G. J. "Propagation of Constraints applied to Circuit Synthesis", *Circuit Theory and Applications*, Vol 8 (1980) pp 127-144.
- [4] Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T. and Berthier, F. "The Constraint Logic Programming Language CHIP", Proceedings, Fifth Generation Computer Systems, Tokyo, December 1988.
- [5] Hansen, B. S. and Hansen, M. R. "Simple Symbolic and Numeric Computations Based on Equations and Inequalities", Computer Science Research Report, IBM Research Laboratory, San Jose, California. June 1985.
- [6] Heintze, N. C., Jaffar, J., Michaylov, S., Stuckey, P. J. and Yap, R. "The CLP(*R*) Programmer's Manual: Version 2.0", Department of Computer Science, Monash University, June 1987.

- [7] Jaffar, J., Michaylov, S., Stuckey, P. J. and Yap, R. "The CLP(R) Language and System", in preparation, 1989.
- [8] Jaffar, J. and Lassez, J.-L. "Constraint Logic Programming", Proceedings, 14th ACM Symposium on POPL, Munich, January 1987.
- [9] Jaffar, J. and Michaylov, S. "Methodology and Implementation of a CLP System", Proceedings, Fourth International Conference on Logic Programming, Melbourne, May 1987.
- [10] Konopasek, M. and Jayaraman, S. "Constraint and Declarative Languages for Engineering Applications: The TK!Solver Contribution" *Proceedings of the IEEE*, Vol. 73, No. 12, December 1985.
- [11] MATHLAB Group, "Macsyma Reference Manual", MIT, 1977.
- [12] Naish, L. "The MU-PROLOG 3.2db Reference Manual", Technical Report, Department of Computer Science, University of Melbourne, 1985.
- [13] Oppenheim, A.V., Willsky, A.S. and Young, I.T. Signals and Systems, Prentice-Hall, 1983.
- [14] Rayna, G. Reduce: Software for Algebraic Computation, Springer-Verlag, New York, 1987.
- [15] Stallman, R. M. and Sussman, G. J. "Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis", AI Memo 380 MIT Artificial Intelligence Laboratory, Cambridge, September 1976. Also in Artificial Intelligence 9 (1977), 135-196.
- [16] Steele, G. L. Jr. and Sussman, G. J. "Constraints", AI Memo 502 MIT Artificial Intelligence Laboratory, Cambridge, November 1978. Invited Paper. Proceedings APL '79. ACM SIGPLAN STAPL APL Quote Quad 9, 4 (June 1979), pp 208-225.
- [17] Steele, G. L. Jr. "The Implementation and Definition of a Computer Programming Language Based on Constraints", Ph.D. Dissertation, Dept. Electrical Engineering and Computer Science. MIT, Cambridge, Mass., Aug. 1980. (MIT-AI TR 595)
- [18] Sussman, G. J. and Stallman, R. M. "Heuristic Techniques in Computer-Aided Circuit Analysis", AI Memo 328, MIT Artificial Intelligence Laboratory, Cambridge, March 1975. Also in *IEEE Transactions on Circuits and Systems* Vol CAS-22 (11) (November 1975).
- [19] Thom, J. and Zobel, J. "NU-Prolog Reference Manual", Version 1.3. Technical Report 86/10. Machine Intelligence Project, Department of Computer Science, University of Melbourne.
- [20] Zima, H. P. "A Constraint Language and its Interpreter", Computer Science Research Report, IBM Research Laboratory, San Jose, California. June 1985.

Appendix I: Circuit Solver

This program carries out a steady state phasor analysis of RLC circuits. It is called through the predicate circuit_solve() which has as arguments

- the angular frequency for the analysis.
- the component list.
- the list of nodes which are to be 'grounded' otherwise all voltages are relative.
- the 'Selection' list a list of nodes for which computed information is to be printed.

The circuit is defined by a list of components. Each component is described by the component type, name, value and the nodes to which it is connected. The component type is used to determine the component characteristics.

```
circuit_solve(W, L, G, Selection) :-
    get_node_vars(L, NV),
    solve(W, L, NV, Handles, G),
    format_print(Handles, Selection).
get_node_vars([[Comp, Num, X, Ns]|Ls], NV) :-
    get_node_vars(Ls, NV1),
    insert_list(Ns, NV1, NV).
get_node_vars([], []).
insert_list([N|Ns], NV1, NV3) :-
    insert_list(Ns, NV1, NV2),
    insert(N, NV2, NV3).
insert_list([], NV, NV).
insert(N, [[N, V, I]|NV1], [[N, V, I]|NV1]).
insert(N, [[N1, V, I] | NV1], [[N1, V, I] | NV2]) :-
    insert(N, NV1, NV2).
insert(N, [], [[N, V, c(0, 0)]]).
solve(W, [X|Xs], NV, [H|Hs], G) :-
    addcomp(W, X, NV, NV1, H),
    solve(W, Xs, NV1, Hs, G).
solve(W, [], NV, [], G) :-
    zero_currents(NV),
    ground_nodes(NV, G).
zero_currents([[N, V, c(0, 0)]|Ls]) :-
    zero_currents(Ls).
zero_currents([]).
ground_nodes(Vs, [N|Ns]) :-
    ground_node(Vs, N),
    ground_nodes(Vs, Ns).
ground_nodes(Vs, []).
ground_node([[N, c(0, 0), I]|Vs], N).
ground_node([[N1, V, I]|Vs], N) :-
    ground_node(Vs, N).
```

```
% The following rules deal with two-terminal components.
%
addcomp(W, [Comp2, Num, X, [N1, N2]], NV, NV2,
           [Comp2, Num, X, [N1, V1, I1], [N2, V2, I2]]):-
    c_neg(I1, I2),
    iv_reln(Comp2, I1, V, X, W),
    c_add(V, V2, V1),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2).
%
% Voltage/current relationships for two-terminal components.
%
iv_reln(resistor, I, V, R, W) :-
    c_mult(I, c(R, 0), V).
iv_reln(voltage_source, I, V, V, W).
iv_reln(isource, I, V, I, W).
iv_reln(capacitor, I, V, C, W) :-
    c_mult(c(0, W*C), V, I).
iv_reln(inductor, I, V, L, W) :-
    c_mult(c(0, W*L), I, V).
iv_reln(connection, I, c(0, 0), L, W).
iv_reln(open, c(0, 0), V, L, W).
iv_reln(diode, I, V, D, W) :-
    diode(D, I, V).
%
% Three rules per diode type.
%
diode(in914, c(I, 0), c(V, 0)) :-
    V < -100, DV = V + 100, I = 10*DV.
diode(in914, c(I, 0), c(V, 0)) :-
    V >= -100, V < 0.6, I = 0.001 * V.
diode(in914, c(I, 0), c(V, 0)) :-
    V >= 0.6, DV = V - 0.6, I = 100 * DV.
%
\% The following rules deal with transistors.
%
addcomp(W, [transistor, Num, X, [N1, N2, N3]], NV, NV3,
           [transistor, Num, X, [N1, V1, I1],
                        [N2, V2, I2], [N3, V3, I3]]):-
    transistor(X, R, Gain),
    c_add(I1, I3, IT),
    c_neg(I2, IT),
    c_add(Vin, V2, V1),
    c_mult(I1, c(R, 0), Vin),
    c_mult(I1, c(Gain, 0), I3),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
```

```
subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    subst([N3, V3, Iold3], [N3, V3, Inew3], NV2, NV3),
    subst([N4, V4, Iold4], [N4, V4, Inew4], NV3, NV4),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2),
    c_add(I3, Iold3, Inew3),
    c_add(I4, Iold4, Inew4).
\% We need one fact for each kind of transistor we wish to consider.
transistor(bc108, 1000, 100).
%
% The following rule deals with transformers.
%
addcomp(W, [transformer, Num, X, [N1, N2, N3, N4]], NV, NV4,
           [transformer, Num, X, [N1, V1, I1], [N2, V2, I2],
           [N3, V3, I3], [N4, V4, I4]]):-
    c_neg(I1, I2),
    c_neg(I3, I4),
    c_add(Vin, V2, V1),
    c_add(Vout, V4, V3),
    c_mult(Vout, c(X, 0), Vin),
    c_mult(I1, c(X, 0), I4),
    subst([N1, V1, Iold1], [N1, V1, Inew1], NV, NV1),
    subst([N2, V2, Iold2], [N2, V2, Inew2], NV1, NV2),
    subst([N3, V3, Iold3], [N3, V3, Inew3], NV2, NV3),
    subst([N4, V4, Iold4], [N4, V4, Inew4], NV3, NV4),
    c_add(I1, Iold1, Inew1),
    c_add(I2, Iold2, Inew2),
    c_add(I3, Iold3, Inew3),
    c_add(I4, Iold4, Inew4).
subst(X, Y, [X|L1], [Y|L1]).
subst(X, Y, [Z|L1], [Z|L2]) :-
    subst(X, Y, L1, L2).
% These rules define complex arithmetic.
%
c_mult(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
    Re3 = Re1 * Re2 + -1 * Im1 * Im2,
    Im3 = Re1*Im2 + Re2*Im1.
c_add(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
    Re3 = Re1 + Re2,
    Im3 = Im1 + Im2.
c_neg(c(Re, Im), c(Re1, Im1)) :-
   Re1 = -Re, Im1 = -Im.
```

```
c_eq(c(Re1, Im1), c(Re2, Im2)) :-
    Re1 = Re2, Im1 = Im2 .
c_real(c(Re, Im), Re).
c_imag(c(Re, Im), Im).
% format_print(H, Selection) --- Print out node information.
```

Appendix II: Transistor Circuit Analysis and Design

This program solves transistor amplifier design and analysis problems, and D.C. circuit analysis problems involving transistors, diodes and capacitors. The first main goal, used for dc analysis of circuits, is

```
?- dc_analysis(Vcc1, Vcc2, Circuit).
```

The parameters are two (optional) source voltages which will ensure that the node cc1 is at voltage Vcc1, similarly for cc2, and a circuit description in the following form: a list of elements each of which is a list containing four elements, the component type, the component name, the data for the component, and a list of nodes that the component connects.

The second main goal, used for transistor amplifier design and analysis, is

```
?- full_analysis(Vcc1, Vcc2, Circuit, In, Out, Type,
Stability, Gain, Inresist, Outresist).
```

The first three parameters are as above. The rest are the input node for the amplifier, the output node for the amplifier, and the type (emitter-follower, common-base) of the amplifier. The final four parameters are design parameters: the stability of collector currents on 50% deviation of Beta values and 10% deviation of *Vbe* values for the transistors of the amplifier, the gain of the amplifier, and finally the open-circuit input resistance and output resistance of the amplifier.

Note that several conventions are followed in this program. In the D.C. case, capacitors are considered to be open circuits. However, for small signal analysis, they are considered to be short circuits. Additionally, it is assumed that any amplifier circuit type used for a full analysis appears in the database of circuits and that all components used appear in the database of components.

```
%
% Entry Points.
%
dc_analysis(Vcc1, Vcc2, Circuit):-
    choose_circuit(Circuit),
    solve_dc(mean, Circuit, [n(cc1, Vcc1, [_]),
        n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
        Nodelist, Collector_Currents),
    current_solve(Nodelist),
    print_circuit(Circuit),
    print_value(Nodelist).
```

```
full_analysis(Vcc1, Vcc2, Circuit, In, Out, Type,
                   Stability, Gain, Inresist, Outresist):-
      % Choose a circuit template.
      circuit(Vcc1, Vcc2, Circuit, In, Out, Type),
      % Construct circuit constraints.
      solve_dc(mean, Circuit, [n(cc1, Vcc1, [_]),
            n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
            Nodelist, Collector_Currents),
      current_solve(Nodelist),
      % Determine stability constraints.
      stability(Vcc1, Vcc2, Circuit, Collector_Currents, Stability),
      % Determine input resistance and gain constraints.
      solve_ss(Circuit, Collector_Currents,
            [n(cc1, 0, [_]), n(cc2, 0, [_]),
             n(gnd, 0, [_]), n(In, 1, [Iin]),
            n(Out, Vout, [])], Nodelist2),
      current_solve(Nodelist2),
      Inresist = -1 / Iin,
      Gain = Vout,
      % Determine Output resistance constraints
      solve_ss(Circuit, Collector_Currents,
            [n(cc1, 0, [_]), n(cc2, 0, [_]),
             n(gnd, 0, [_]), n(Out, 1, [Iout])], Nodelist3),
      current_solve(Nodelist3),
      Outresist = -1 / Iout,
      % Choose circuit values - all (real) choice points occur here
      choose_circuit(Circuit).
% Small signal equivalent circuit analysis.
%
solve_ss([], [], List, List).
solve_ss([[Component, _, Data, Points]|Rest], CCin, Innodes, Outnodes):-
      connecting(Points, Volts, Amps, Innodes, Tmpnodes),
      component_ss(Component, Data, Volts, Amps, CCin, CCout),
      solve_ss(Rest, CCout, Tmpnodes, Outnodes).
component_ss(resistor, R, [V1, V2], [I, -1*I], Cc, Cc):-
      V1-V2 = R*I.
component_ss(capacitor, _, [V, V], [I, -1*I], Cc, Cc).
component_ss(transistor, [npn, Code, active, Mean, _, _],
        [Vb, Vc, Ve], [Ib, Ic, Ie], [Icol|CC], CC):-
      Mean = data(Beta, _, _, Vt),
      Vb - Ve = (Beta*Vt / Icol)*Ib,
      Ic = Beta*Ib,
      Ie + Ic + Ib = 0.
% D.C. component solving.
```

```
%
solve_dc(_, [], List, List, []).
solve_dc(Kind, [[Component, _, Data, Points] | Rest], Inlist, Outlist, CCin):-
      connecting(Points, Volts, Amps, Inlist, Tmplist),
      component_dc(Component, Data, Volts, Amps, CCin, CCout, Kind),
      solve_dc(Kind, Rest, Tmplist, Outlist, CCout).
component_dc(resistor, R, [V1, V2], [I, -1*I], Cc, Cc, _):-
      V1-V2 = R*I.
component_dc(capacitor, _, [V1, V2], [0, 0], Cc, Cc, _).
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, mean):-
      Mean = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts, [Ib, Ic, Ie]
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, minn):-
      Min = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts, [Ib, Ic, Ie])
component_dc(transistor, [Type, Code, State, Mean, Min, Max],
        Volts, [Ib, Ic, Ie], [Ic|CC], CC, maxx):-
      Max = data(Beta, Vbe, Vcestat, _),
      transistor_state(Type, State, Beta, Vbe, Vcesat, Volts, [Ib, Ic, Ie])
component_dc(diode, [Code, State, Vf, Vbreak], Volts, Amps, Cc, Cc, _):-
      diode_state(State, Vf, Vreak, Volts, Amps).
%
% Diode and transistor states / relationships.
%
diode_state(forward, Vf, Vbreak, [Vp, Vm], [I, -1*I]):-
      % forward biased
      Vp - Vm = Vf,
      I >= 0.
diode_state(reverse, Vf, Vbreak, [Vp, Vm], [I, -1*I]):-
      % reverse biased
      Vp - Vm < Vf,
      Vm - Vp < Vbreak,
      I = 0.
transistor_state(npn, active, Beta, Vbe, _, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc >= Vb,
      Ib >= 0,
      Ic = Beta*Ib,
      Ie+Ib+Ic = 0.
transistor_state(pnp, active, Beta, Vbe, _, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc <= Vb,
      Ib <= 0,
      Ic = Beta*Ib,
      Ie+Ib+Ic = 0.
```

```
transistor_state(npn, saturated, Beta, Vbe,
                      Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc = Ve + Vcesat,
      Ib >= 0,
      Ic >= 0,
      Ie+Ib+Ic = 0.
transistor_state(pnp, saturated, Beta, Vbe,
                      Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb = Ve + Vbe,
      Vc = Ve + Vcesat,
      Ib <= 0,
      Ic <= 0,
      Ie+Ib+Ic = 0.
transistor_state(npn, cutoff, Beta, Vbe,
                      Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb <= Ve + Vbe,
      Ib = 0,
      Ic = 0,
      Ie = 0.
transistor_state(pnp, cutoff, Beta, Vbe,
                      Vcesat, [Vb, Vc, Ve], [Ib, Ic, Ie]):-
      Vb >= Ve + Vbe,
      Ib = 0,
      Ic = 0,
      Ie = 0.
%
% Component connections.
%
connecting([], [], [], List, List).
connecting([P|PR], [V|VR], [I|IR], Inlist, Outlist):-
      connect(P, V, I, Inlist, Tmplist),
      connecting(PR, VR, IR, Tmplist, Outlist).
connect(P, V, I, [], [n(P, V, [I])]):-!.
connect(P, V, I, [n(P, V, Ilist) | Rest], [n(P, V, [I|Ilist])|Rest]):-!.
connect(P, V, I, [A|Rest], [A|Newrest]) :-
      connect(P, V, I, Rest, Newrest).
%
% Stability analysis.
%
stability(Vcc1, Vcc2, Circuit, CollectorCurrents, Stability):-
      solve_dc(minn, Circuit, [n(cc1, Vcc1, [_]),
          n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
          Nodelist1, MinCurrents),
      current_solve(Nodelist1),
      solve_dc(maxx, Circuit , [n(cc1, Vcc1, [_]),
          n(cc2, Vcc2, [_]), n(gnd, 0, [_])],
          Nodelist2, MaxCurrents),
```

```
current_solve(Nodelist2),
      calculate(MinCurrents, MaxCurrents, CollectorCurrents, Stability).
calculate(MinCurrents, MaxCurrents, CollectorCurrents, Stability):-
      cal(MinCurrents, MaxCurrents, CollectorCurrents, Percents),
      maxi(Percents, 0, Stability).
cal([Min|Rin], [Max|Rax], [Ic|Rc], [Pc|Rpc]):-
      Pc = max(Ic-Min, Max-Ic),
      cal(Rin, Rax, Rc, Rpc).
cal([], [], [], []).
maxi([N1|R], N2, P):-
      M = max(N1, N2),
      maxi(R, M, P).
maxi([], P, P).
current_solve([]).
current_solve([n(_, _, L) | Rest]) :-
      kcl(L),
      current_solve(Rest).
print_value([]).
print_value([n(P, V, I) | Rest]) :-
      printf("% at % %\n", [P, V, I]),
      print_value(Rest).
print_circuit([]).
print_circuit([[Comp, Name, Data, Points] | Rest]) :-
      printf(" % at % %\n", [Comp, Name, Data]),
      print_circuit(Rest).
sum([X|T], Z) :-
      X+P = Z,
      sum(T, P).
sum([], 0).
kcl(L) :-
      sum(L, 0).
%
% Choose circuit values.
%
choose_circuit([[Component_type, _, Data, _] | RestofCircuit]):-
      choose_component(Component_type, Data),
      choose circuit(RestofCircuit).
choose circuit([]).
choose_component(resistor, R):-
      resistor_val(R).
```

```
choose_component(capacitor, _).
choose_component(diode, [Code, _, Vf, Vbreak]):- diode_type(Code, Vf, Vbreak).
choose_component(transistor, [Type, Code, _, Mean, Min, Max]):-
      transistor_type(Type, Code, MeanBeta,
                           MeanVbe, MeanVcestat, MeanVt, mean),
      Mean = data(MeanBeta, MeanVbe, MeanVcestat, MeanVt),
      transistor_type(Type, Code, MinBeta,
                           MinVbe, MinVcestat, MinVt, minn),
      Min = data(MinBeta, MinVbe, MinVcestat, MinVt),
      transistor_type(Type, Code, MaxBeta,
                           MaxVbe, MaxVcestat, MaxVt, maxx),
      Max = data(MaxBeta, MaxVbe, MaxVcestat, MaxVt).
%
\% Database of circuits and components.
%
resistor_val(100).
resistor_val(50).
resistor_val(27).
resistor_val(10).
resistor_val(5).
resistor_val(2).
resistor_val(1).
diode_type(di1, 0.6, 100).
transistor_type(npn, tr0, 100, 0.7, 0.3, 0.025, mean).
transistor_type(npn, tr0, 50, 0.8, 0.3, 0.025, minn).
transistor_type(npn, tr0, 150, 0.6, 0.3, 0.025, maxx).
transistor_type(pnp, tr1, 100, -0.7, -0.3, 0.025, mean).
transistor_type(pnp, tr1, 50, -0.8, -0.3, 0.025, minn).
transistor_type(pnp, tr1, 150, -0.6, -0.3, 0.025, maxx).
circuit(15, 0, [
    [capacitor, c1, c1, [in, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
    [transistor, tr, [npn, tr0, active, Mean, Minn, Maxx], [b, c, e]],
    [resistor, re, Re, [e, gnd]],
    [capacitor, c2, c2, [c, out]],
    [resistor, rc, Rc, [c, cc1]],
    [capacitor, c3, c3, [e, gnd]]],
  in, out, common_emitter).
circuit(15, 0, [
    [capacitor, c1, C1, [gnd, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
    [transistor, tr, [pnp, tr1, active, Mean, Minn, Maxx], [b, c, e]],
    [resistor, re, Re, [e, gnd]],
```

```
[capacitor, c2, C2, [c, in]],
    [resistor, rc, Rc, [c, cc1]],
    [capacitor, c3, C3, [e, out]]],
    in, out, common_base).
circuit(15, 0, [
    [capacitor, c1, C1, [in, b]],
    [resistor, r1, R1, [b, cc1]],
    [resistor, r2, R2, [b, gnd]],
    [transistor, tr, [npn, tr0, active, Mean, Minn, Maxx], [b, cc1, e]],
    [resistor, re, Re, [e, gnd]],
    [capacitor, c3, C3, [e, out]]],
    in, out, emitter_follower).
```

Appendix III: Signal Flow Graph Simulation

This program simulates the signal flow graph presented to it as input, and describes the value at the required node at each time interval by drawing a graph. The goal is of the form

?- flow(Spec, Output).

The first argument describes the signal flow graph as a list of arcs and the second is either the name of a node in the graph for which the value is to be plotted, or an empty list signifying that the value at each node is to be printed at each time interval. Each arc description is of the form [delay, node1, node2] or [coeff(coefficient), node1, node2].

```
%
% First convert the goal to an internal representation.
flow(Spec, Output) :-
         convert_list(Spec, Specd, [Nodes, Sources]),
         get_node_index(Output, Nodes, _, Oindex, no), !,
         analyze(Specd, Oindex).
convert_list([S|Ss], Spec2, Info2) :-
         convert_list(Ss, Spec1, Info1),
         convert_arc(S, Spec1, Spec2, Info1, Info2).
convert_list([], [], [[], Sources]) :-
         read_sources(Sources).
convert_arc([Type, In, Out], Spec1, Spec4, Info1, Info3) :-
         get_index(Type, In, Info1, Info2, Index1, Newnode_flag1),
         get_index(not_source, Out, Info2, Info3, Index2, Newnode_flag2),
         node_insert(Spec1, Spec2, Newnode_flag1),
         node_insert(Spec2, Spec3, Newnode_flag2),
         insert(Spec3, Spec4, Index2, [Type, Index1]).
get_index(source, N, [Nodes1, Sources], [Nodes1, Sources], Index, no) :-
```

```
get_node_index(N, Sources, _, Index, no).
get_index(Type, Node, [Nodes1, Sources], [Nodes2, Sources],
           Index, Newnode_flag) :-
         get_node_index(Node, Nodes1, Nodes2, Index, Newnode_flag).
%
% Find an index to a node in the list, adding it if necessary.
%
% get_node_index(Node, Old_node_list, New_node_list, Index, Newnode_flag)
%
get_node_index(X, [X|Ns], [X|Ns], 1.0, no).
get_node_index(X, [Xd|Ns], [Xd|Nds], V + 1, Newnode_flag) :-
         get_node_index(X, Ns, Nds, V, Newnode_flag).
get_node_index(X, [], [X], 1, yes).
%
\% Add new nodes by setting up a new list on the specification list.
node_insert(S, S, no).
node_insert([S1|Ss1], [S1|Ss2], yes) :-
         node_insert(Ss1, Ss2, yes).
node_insert([], [[]], yes).
%
% Insert the arc in the new specification.
%
insert([S|Ss], [S|Ts], Index, Arc) :-
         Index > 1,
         insert(Ss, Ts, Index-1, Arc).
insert([S|Ss], [[Arc|S]|Ss], 1, Arc).
analyze(Spec, Output) :-
         getinitial(Old),
         sigflow(Old, Spec, Output).
sigflow(Old, Spec, Output) :-
         getsources(Sources),
         stepflow(Old, Sources, New, Spec, New),
         printnodes(New, Output),
         sigflow(New, Spec, Output).
getsources(Ss) :-
         readsources(Ss).
getsources(0, []).
getinitial(Old) :-
         readinitial(Old).
%
% stepflow(Oldnodes, Sources, Newnodes, Spec, Newnodes_left_to_process)
%
```

```
stepflow(Oldnodes, Sources, Newnodes, [S|Spec], [N|New]) :-
         stepflow(Oldnodes, Sources, Newnodes, Spec, New),
         calcnode(Oldnodes, Sources, Newnodes, S, N).
stepflow(Oldnodes, Sources, Newnodes, [], []).
% calcnode(Oldnodes, Sources, Newnodes, Arcs, Newnode)
calcnode(Oldnodes, Sources, Newnodes, [Arc|Arcs], New) :-
         New = New1 + New2,
         calcarc(Oldnodes, Sources, Newnodes, Arc, New1),
         calcnode(Oldnodes, Sources, Newnodes, Arcs, New2).
calcnode(Oldnodes, Sources, Newnodes, [], 0).
%
% calcarc(Oldnodes, Sources, Newnodes, Arc, New)
%
calcarc(Oldnodes, Sources, Newnodes, [coeff(C), Arc], New) :-
         New - C*Value = 0,
         find_index(Value, Newnodes, Arc).
calcarc(Oldnodes, Sources, Newnodes, [delay, NIndex], New) :-
         New - Value = 0,
         find_index(Value, Oldnodes, NIndex).
calcarc(Oldnodes, Sources, Newnodes, [source, Name], New) :-
         New - Value = 0,
         find_index(Value, Sources, Name).
%
% find_index(item, list of items, place in list)
%
find_index(V, [V|Vs], 1).
find_index(VV, [V|Vs], N) :-
         N > 0,
         NN = N-1,
         find_index(VV, Vs, NN).
% printnodes() - print out output signal
% readsources(), readinital() - read data from files
```